

775 2nd Wk.

2

AD-A205 908

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

88-05-31-RAT

Rational R1000 Series 200 Model 20 and VAX-11/750

17 August 1988

Wright-Patterson AFB OH 45433-6503

Washington DC 20301-3081

2000	✓
2001	
2002	
2003	
2004	
2005	
2006	
2007	
2008	
2009	
2010	
2011	
2012	
2013	
2014	
2015	
2016	
2017	
2018	
2019	
2020	
2021	
2022	
2023	
2024	
2025	
2026	
2027	
2028	
2029	
2030	

Ada Compiler Validation Summary Report:

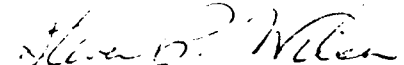
Compiler Name: VAX\_VMS, Version 2.0.45


Certificate Number: 880815W1.09143

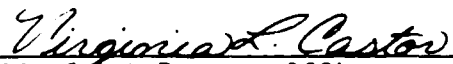
Host:	Target:
Rational R1000 Series 200	VAX-11/750 under
Model 20 under D_10_9_10wps	VMS, Version 4.5

Testing Completed 17 August 1988 Using ACVC 1.9

This report has been reviewed and is approved.

  
\_\_\_\_\_  
Ada Validation Facility  
Steven P. Wilson  
Technical Director  
ASD/SCEL  
Wright-Patterson AFB OH 45433-6503

  
\_\_\_\_\_  
Ada Validation Organization  
Dr. John F. Kramer  
Institute for Defense Analyses  
Alexandria VA 22311

  
\_\_\_\_\_  
Ada Joint Program Office  
Virginia L. Castor  
Director  
Department of Defense  
Washington DC 20301

Ada Compiler Validation Summary Report:

Compiler Name: VAX\_VMS, Version 2.0.45

Certificate Number: 880815W1.09143

Host:

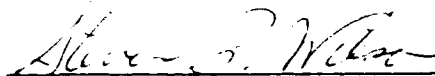
Rational R1000 Series 200  
Model 20 under D\_10\_9\_10wps

Target:

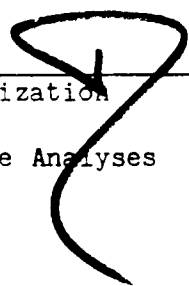
VAX-11/750 under  
VMS, Version 4.5

Testing Completed 17 August 1988 Using ACVC 1.9

This report has been reviewed and is approved.



Ada Validation Facility  
Steven P. Wilson  
Technical Director  
ASD/SCEL  
Wright-Patterson AFB OH 45433-6503



Ada Validation Organization  
Dr. John F. Kramer  
Institute for Defense Analyses  
Alexandria VA 22311

Ada Joint Program Office  
Virginia L. Castor  
Director  
Department of Defense  
Washington DC 20301

## TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION	
1.1	PURPOSE OF THIS VALIDATION SUMMARY REPORT . . . . .	1-2
1.2	USE OF THIS VALIDATION SUMMARY REPORT . . . . .	1-2
1.3	REFERENCES . . . . .	1-3
1.4	DEFINITION OF TERMS . . . . .	1-3
1.5	ACVC TEST CLASSES . . . . .	1-4
CHAPTER 2	CONFIGURATION INFORMATION	
2.1	CONFIGURATION TESTED . . . . .	2-1
2.2	IMPLEMENTATION CHARACTERISTICS . . . . .	2-2
CHAPTER 3	TEST INFORMATION	
3.1	TEST RESULTS . . . . .	3-1
3.2	SUMMARY OF TEST RESULTS BY CLASS . . . . .	3-1
3.3	SUMMARY OF TEST RESULTS BY CHAPTER . . . . .	3-2
3.4	WITHDRAWN TESTS . . . . .	3-2
3.5	INAPPLICABLE TESTS . . . . .	3-2
3.6	TEST, PROCESSING, AND EVALUATION MODIFICATIONS . . . . .	3-4
3.7	ADDITIONAL TESTING INFORMATION . . . . .	3-5
3.7.1	Prevalidation . . . . .	3-5
3.7.2	Test Method . . . . .	3-5
3.7.3	Test Site . . . . .	3-6
APPENDIX A	DECLARATION OF CONFORMANCE	
APPENDIX B	APPENDIX F OF THE Ada STANDARD	
APPENDIX C	TEST PARAMETERS	
APPENDIX D	WITHDRAWN TESTS	

## CHAPTER 1

### INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent but permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

## INTRODUCTION

### 1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- . To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
- . To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard
- . To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by SofTech, Inc. under the direction of the AVF according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO). On-site testing was completed 17 August 1988 at Rational, Santa Clara CA.

### 1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse  
Ada Joint Program Office  
OUSDRE  
The Pentagon, Rm 3D-139 (Fern Street)  
Washington DC 20301-3081

or from:

Ada Validation Facility  
ASD/SCEL  
Wright-Patterson AFB OH 45433-6503

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization  
Institute for Defense Analyses  
1801 North Beauregard Street  
Alexandria VA 22311

## 1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
2. Ada Compiler Validation Procedures and Guidelines, Ada Joint Program Office, 1 January 1987.
3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1986.
4. Ada Compiler Validation Capability User's Guide, December 1986.

## 1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.
Ada Commentary	An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.
Ada Standard	ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the <u>Ada Compiler Validation Procedures and Guidelines</u> .
AVO	The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical



## INTRODUCTION

support for Ada validations to ensure consistent practices.

Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.
Failed test	An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.
Inapplicable test	An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Passed test	An ACVC test for which a compiler generates the expected result.
Target	The computer for which a compiler generates code.
Test	A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.
Withdrawn test	An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

### 1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce compilation or link errors.

Class A tests check that legal Ada programs can be successfully compiled and executed. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated.

Two library units, the package REPORT and the procedure CHECK\_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK\_FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard. The operation of REPORT and CHECK\_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of the tests in the ACVC follow conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and

## INTRODUCTION

place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

## CHAPTER 2

### CONFIGURATION INFORMATION

#### 2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: VAX\_VMS, Version 2.0.45

ACVC Version: 1.9

Certificate Number: 880815W1.09143

Host Computer:

Machine:	Rational R1000 Series 200 Model 20
----------	---------------------------------------

Operating System:	D_10_9_10wps
-------------------	--------------

Memory Size:	32 Megabytes
--------------	--------------

Target Computer:

Machine:	VAX-11/750
----------	------------

Operating System:	VMS, Version 4.5
-------------------	------------------

Memory Size:	6 Megabytes
--------------	-------------

Communications Network:	TCP/IP/ETHERNET
-------------------------	-----------------

## CONFIGURATION INFORMATION

### 2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

- . Capacities.

The compiler correctly processes tests containing loop statements nested to 65 levels, block statements nested to 65 levels, and recursive procedures separately compiled as subunits nested to 10 levels. It correctly processes a compilation containing 723 variables in the same declarative part. (See tests D55A03A..H (8 tests), D56001B, D64005E..G (3 tests), and D29002K.)

- . Universal integer calculations.

An implementation is allowed to reject universal integer calculations having values that exceed `SYSTEM.MAX_INT`. This implementation processes 64 bit integer calculations. (See tests D4A002A, D4A002B, D4A004A, and D4A004B.)

- . Predefined types.

This implementation supports the additional predefined types `SHORT_INTEGER`, `LONG_FLOAT`, and `SHORT_SHORT_INTEGER` in the package `STANDARD`. (See tests B86001C and B86001D.)

- . Based literals.

An implementation is allowed to reject a based literal with a value exceeding `SYSTEM.MAX_INT` during compilation, or it may raise `NUMERIC_ERROR` or `CONSTRAINT_ERROR` during execution. This implementation rejects the test during compilation. (See test E24101A.)

- . Expression evaluation.

Apparently no default initialization expressions for record components are evaluated before any value is checked to belong to a component's subtype. (See test C32117A.)

Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

This implementation uses no extra bits for extra precision. This implementation uses all extra bits for extra range. (See test C35903A.)

Sometimes `NUMERIC_ERROR` is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

Apparently `NUMERIC_ERROR` is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

Apparently underflow is not gradual. (See tests C45524A..Z.)

. Rounding.

The method used for rounding to integer is apparently round away from zero. (See tests C46012A..Z.)

The method used for rounding to longest integer is apparently round away from zero. (See tests C46012A..Z.)

The method used for rounding to integer in static universal real expressions is apparently round away from zero. (See test C4A014A.)

. Array types.

An implementation is allowed to raise `NUMERIC_ERROR` or `CONSTRAINT_ERROR` for an array having a `'LENGTH` that exceeds `STANDARD.INTEGER'LAST` and/or `SYSTEM.MAX_INT`. For this implementation:

Declaration of an array type or subtype declaration with more than `SYSTEM.MAX_INT` components raises `NUMERIC_ERROR`. (See test C36003A.)

`NUMERIC_ERROR` is raised when an array type with `INTEGER'LAST + 2` components is declared. (See test C36202A.)

`NUMERIC_ERROR` is raised when an array type with `SYSTEM.MAX_INT + 2` components is declared. (See test C36202B.)

A packed `BOOLEAN` array having a `'LENGTH` exceeding `INTEGER'LAST` raises `NUMERIC_ERROR` when the array type is declared. (See test C52103X.)

A packed two-dimensional `BOOLEAN` array with more than `INTEGER'LAST` components raises `NUMERIC_ERROR` when the array type is declared. (See test C52104Y.)

## CONFIGURATION INFORMATION

A null array with one dimension of length greater than `INTEGER'LAST` may raise `NUMERIC_ERROR` or `CONSTRAINT_ERROR` either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises `NUMERIC_ERROR` when the array type is declared. (See test E52103Y.)

In assigning one-dimensional array types, the expression appears to be evaluated in its entirety before `CONSTRAINT_ERROR` is raised when checking whether the expression's subtype is compatible with the target's subtype. In assigning two-dimensional array types, the expression does not appear to be evaluated in its entirety before `CONSTRAINT_ERROR` is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

### . Discriminated types.

During compilation, an implementation is allowed to either accept or reject an incomplete type with discriminants that is used in an access type definition with a compatible discriminant constraint. This implementation accepts such subtype indications. (See test E38104A.)

In assigning record types with discriminants, the expression appears to be evaluated in its entirety before `CONSTRAINT_ERROR` is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

### . Aggregates.

In the evaluation of a multi-dimensional aggregate, all choices appear to be evaluated before checking against the index type. (See tests C43207A and C43207B.)

In the evaluation of an aggregate containing subaggregates, not all choices are evaluated before being checked for identical bounds. (See test E43212B.)

`CONSTRAINT_ERROR` is raised before all choices are evaluated when a bound in a nonnull range of a nonnull aggregate does not belong to an index subtype. (See test E43211B.)

### . Representation clauses.

An implementation might legitimately place restrictions on representation clauses used by some of the tests. If a representation clause is used by a test in a way that violates a restriction, then the implementation must reject it.

For this implementation:

Enumeration representation clauses containing noncontiguous values for enumeration types other than character and boolean types are supported. (See tests C35502I..J, C35502M..N, and A39005F.)

Enumeration representation clauses containing noncontiguous values for character types are supported. (See tests C35507I..J, C35507M..N, and C55B16A.)

Enumeration representation clauses for derived types are not supported. (See tests C35508I..J and C35508M..N.)

Length clauses with SIZE specifications for enumeration types are supported. (See test A39005B.)

Length clauses with STORAGE\_SIZE specifications for access types are supported. (See tests A39005C and C87B62B.)

Length clauses with STORAGE\_SIZE specifications for task types are supported. (See tests A39005D and C87B62D.)

Length clauses with SMALL specifications are supported. (See tests A39005E and C87B62C.)

Record representation clauses are supported. (See test A39005G.)

Length clauses with SIZE specifications for derived integer types are not supported. (See test C87B62A.)

. Pragma.

The pragma `INLINE` is not supported for procedures or functions. (See tests LA3004A, LA3004B, EA3004C, EA3004D, CA3004E, and CA3004F.)

. Input/output.

The package `SEQUENTIAL_IO` can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

The package `DIRECT_IO` cannot be instantiated with unconstrained array types or record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

Modes `IN_FILE` and `OUT_FILE` are supported for `SEQUENTIAL_IO`. (See tests CE2102D and CE2102E.)



## CONFIGURATION INFORMATION

Modes `IN_FILE`, `OUT_FILE`, and `INOUT_FILE` are supported for `DIRECT_IO`. (See tests CE2102F, CE2102I, and CE2102J.)

`RESET` and `DELETE` are supported for `SEQUENTIAL_IO` and `DIRECT_IO`. (See tests CE2102G and CE2102K.)

Dynamic creation and deletion of files are supported for `SEQUENTIAL_IO` and `DIRECT_IO`. (See tests CE2106A and CE2106B.)

Overwriting to a sequential file truncates the file to last element written. (See test CE2208B.)

An existing text file can be opened in `OUT_FILE` mode, can be created in `OUT_FILE` mode, and cannot be created in `IN_FILE` mode. (See test EE3102C.)

More than one internal file can be associated with each external file for text Input/output for reading only. (See tests CE3111A..E (5 tests), CE3114B, and CE3115A.)

More than one internal file can be associated with each external file for sequential Input/output for reading only. (See tests CE2107A..D (4 tests), CE2110B, and CE2111D.)

More than one internal file can be associated with each external file for direct Input/output for reading only. (See tests CE2107F..I (5 tests), CE2110B, and CE2111H.)

An internal sequential access file and an internal direct access file cannot be associated with a single external file for writing. (See test CE2107E.)

Temporary sequential files and temporary direct files are given names. Temporary files given names are deleted when they are closed. (See tests CE2108A and CE2108C.)

### . Generics.

Generic subprogram declarations and bodies can be compiled in separate compilations. (See tests CA1012A and CA2009F.)

Generic package declarations and bodies can be compiled in separate compilations. (See tests CA2009C, BC3204C, and BC3205D.)

Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

## CHAPTER 3

### TEST INFORMATION

#### 3.1 TEST RESULTS

Version 1.9 of the ACVC comprises 3122 tests. When this compiler was tested, 27 tests had been withdrawn because of test errors. The AVF determined that 345 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 285 executable tests that use floating-point precision exceeding that supported by the implementation. Modifications to the code, processing, or grading for 76 tests were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

#### 3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	108	1049	1520	15	14	44	2750
Inapplicable	2	2	333	2	4	2	345
Withdrawn	3	2	21	0	1	0	27
TOTAL	113	1053	1874	17	19	46	3122

## TEST INFORMATION

### 3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER														TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14		
Passed	184	465	490	245	165	98	141	326	131	36	23	3	232	2750	
Inapplicable	20	107	184	3	1	0	2	1	6	0	0	0	21	345	
Withdrawn	2	14	3	0	0	1	2	0	0	0	2	1	2	27	
TOTAL	206	586	677	248	166	99	145	327	137	36	23	4	255	3122	

### 3.4 WITHDRAWN TESTS

The following 27 tests were withdrawn from ACVC Version 1.9 at the time of this validation:

B28003A	E28005C	C34004A	C35502P	A35902C
C35904A	C35904B	C35A03E	C35A03R	C37213H
C37213J	C37215C	C37215E	C37215G	C37215H
C38102C	C41402A	C45332A	C45614C	A74106C
C85018B	C87B04B	CC1311B	BC3105A	AD1A01A
CE2401H	CE3208A			

See Appendix D for the reason that each of these tests was withdrawn.

### 3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 345 tests were inapplicable for the reasons indicated:

- C35508I..J (2 tests) and C35508M..N (2 tests) use enumeration representation clauses for derived types. These clauses are not supported by this compiler.
- C35702A uses SHORT\_FLOAT which is not supported by this implementation.

## TEST INFORMATION

- . A39005G uses a record representation clause which attempts to pack a record component that is of an array subtype, and such a record representation clause is not supported by this compiler.
- . The following tests use LONG\_INTEGER, which is not supported by this compiler:

C45231C	C45304C	C45502C	C45503C	C45504C
C45504F	C45611C	C45613C	C45631C	C45632C
B52004D	B55B09C	C55B07A		

- . C45531M, C45531N, C45532M, and C45532N use fine 48-bit fixed-point base types which are not supported by this compiler.
- . C455310, C45531P, C455320, and C45532P use coarse 48-bit fixed-point base types which are not supported by this compiler.
- . C4A013B uses a static value that is outside the range of the most accurate floating-point base type. The declaration was rejected at compile time.
- . D4A004B uses a numeric literal greater than SYSTEM.MAX\_INT which is not supported by this compiler.
- . D64005G uses nested procedures as subunits to a level of 17, which exceeds the capacity of the compiler.
- . C86001F redefines package SYSTEM, but TEXT\_IO is made obsolete by this new definition in this implementation and the test cannot be executed since the package REPORT is dependent on the package TEXT\_IO.
- . C87B62A uses length clauses with SIZE specifications for derived integer types which are not supported by this compiler.
- . C96005B requires the range of type DURATION to be different from those of its base type; in this implementation they are the same.
- . CA3004E, EA3004C, and LA3004A use the INLINE pragma for procedures, which is not supported by this compiler.
- . CA3004F, EA3004D, and LA3004B use the INLINE pragma for functions, which is not supported by this compiler.
- . AE2101H, EE2401D, and EE2401G use instantiations of package DIRECT\_IO with unconstrained array types and record types having discriminants without defaults. These instantiations are rejected by this compiler.
- . CE2105A..B (2 tests) and CE3109A attempt to create a file of mode IN\_FILE. Creation of an external file with mode IN\_FILE is not supported by this compiler.

## TEST INFORMATION

- CE2107B..E (4 tests), CE2107G..I (3 tests), CE2110B, CE2111D, CE2111H, CE3111B..E (4 tests), and CE3114B are inapplicable because multiple internal files cannot be associated with the same external file for both read and write and for write only. The proper exception is raised when multiple access is attempted.
- The following 285 tests require a floating-point accuracy that exceeds the maximum of 9 digits supported by this implementation:

C24113F..Y (20 tests)	C35705F..Y (20 tests)
C35706F..Y (20 tests)	C35707F..Y (20 tests)
C35708F..Y (20 tests)	C35802F..Z (21 tests)
C45241F..Y (20 tests)	C45321F..Y (20 tests)
C45421F..Y (20 tests)	C45521F..Z (21 tests)
C45524F..Z (21 tests)	C45621F..Z (21 tests)
C45641F..Y (20 tests)	C46012F..Z (21 tests)

### 3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that wasn't anticipated by the test (such as raising one exception instead of another).

Modifications were required for 76 Class B tests.

The following Class B tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B22003A	B22004A	B22004B	B22004C	B23004A
B23004B	B24001A	B24001B	B24001C	B24005A
B24005B	B24007A	B24009A	B24204A	B24204B
B24204C	B25002B	B26001A	B26002A	B26005A
B28003C	B29001A	B2A003A	B2A003B	B2A003C
B2A007A	B32103A	B33201B	B33202B	B33203B
B33301A	B35101A	B36002A	B36201A	B37201A
B37205A	B37307B	B38003A	B38003B	B38009A
B38009B	B41201A	B41202A	B44001A	B44004B
B44004C	B45205A	B48002A	B48002D	B51001A
B51003A	B51003B	B53003A	B55A01A	B64001A
B64006A	B67001A	B67001B	B67001C	B67001D
B74003A	B91001H	B91003B	B95001A	B95003A

B95004A	B95079A	B97101A	BB3005A	BC1303F
BC2001D	BC2001E	BC3003A	BC3003B	BC3005B
BC3013A				

### 3.7 ADDITIONAL TESTING INFORMATION

#### 3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.9 produced by the VAX\_VMS compiler, Version 2.0.45, was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

#### 3.7.2 Test Method

Testing of the VAX\_VMS compiler, Version 2.0.45, using ACVC Version 1.9 was conducted on-site by a validation team from the AVF. The configuration consisted of a Rational R1000 Series 200 Model 20 host operating under D\_10\_9\_10wps, and a VAX-11/750 target operating under VMS, Version 4.5. The host and target computers were linked via TCP/IP/ETHERNET.

A magnetic tape containing all tests except for withdrawn tests and tests requiring unsupported floating-point precisions was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. Tests requiring modifications during the prevalidation testing were included in their modified form on the magnetic tape.

The contents of the magnetic tape were loaded directly onto the host computer. After the test files were loaded to disk, the full set of tests was compiled and linked on the Rational R1000 Series 200 Model 20, and all executable tests were run on the VAX-11/750. Object files were linked on the host computer, and executable images were transferred to the target computer via TCP/IP/ETHERNET. Results were printed from the host computer, with results being transferred to the host computer via TCP/IP/ETHERNET.

The compiler was tested using command scripts provided by Rational and reviewed by the validation team. The compiler was tested using all default switch settings except for the following:

## TEST INFORMATION

<u>Switch</u>	<u>Effect</u>
Create_Subprogram_Specs = False	Separate subprogram specification is not generated automatically.
Remote_Directory = "sys\$sysroot:[rational. crud.acvc.report]"	Denotes directory on the target into which files should be automatically downloaded.
Remote_Machine = "shemp"	Denotes the VAX target machine.
Optimization_Level = 4	Sets optimization to an intermediate level.

Tests were compiled, linked, and executed (as appropriate) using a single host computer and a single target computer. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

### 3.7.3 Test Site

Testing was conducted at Rational, Santa Clara CA and was completed on 17 August 1988.

## APPENDIX A

### DECLARATION OF CONFORMANCE

Rational has submitted the following Declaration of Conformance concerning the VAX\_VMS compiler, Version 2.0.45.



## DECLARATION OF CONFORMANCE

Compiler Implementor: Rational

Ada® Validation Facility: ASD/SCEL, Wright-Patterson AFB, OH 45433-6503

Ada Compiler Validation Capability (ACVC) Version: 1.9

### Base Configuration

Base Compiler Name: VAX\_VMS

Version: 2.0.45

Host Architecture: R1000®

Operating System: D\_10\_9\_10wps

Target Architecture: DEC VAX

Operating System: VMS 4.5

DEC VAX 11/750

### Implementor's Declaration

I, the undersigned, representing Rational, have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in the compiler listed in this declaration. I declare that Rational is the owner of record of the Ada language compiler listed above and, as such, is responsible for maintaining said compiler in conformance to ANSI/MIL-STD-1815A. All certificates and registrations for the Ada language compiler listed in this declaration shall be made only in the owner's corporate name.

  
Walter A. Wallach, Manager, Software Test

Date: 8/18/88

### Owner's Declaration

I, the undersigned, representing Rational, take full responsibility for the implementation and maintenance of the Ada compiler listed above, and agree to the public disclosure of the final Validation Summary Report. I further agree to continue to comply with the Ada trademark policy, as defined by the Ada Joint Program Office. I declare that all of the Ada language compilers listed, and their host/target performance are in compliance with the Ada Language Standard ANSI/MIL-STD-1815A.

  
Walter A. Wallach, Manager, Software Test

Date: 8/18/88

®Ada is a registered trademark of the United States Government (Ada Joint Program Office).

®R1000 is the registered trademark of Rational.

## APPENDIX B

### APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the VAX\_VMS compiler, Version 2.0.45, are described in the following sections, taken from Appendix F of the Ada Standard. Implementation-specific portions of the package STANDARD are also included in this appendix.

package STANDARD is

...

type INTEGER is range -2147483648 .. 2147483647;

type SHORT\_INTEGER is range -32768 .. 32767;

type SHORT\_SHORT\_INTEGER is range -128 .. 127;

type FLOAT is digits 6 range -1.70141173319264E+38 .. 1.70141173319264E+38;

type LONG\_FLOAT is digits 9 range  
-1.70141183460469E+38 .. 1.70141183460469E+38;

type DURATION is delta 6.10351562500000E-05 range  
1.31072000000000E+05 .. 1.31071999938965E+05;

end STANDARD;

## 1. Implementation-Dependent Pragmas

The Vax/VMS Cross-Compiler supports pragmas for application software development in addition to those listed in Appendix B of the *Reference Manual for the Ada Programming Language*. They are described below, along with additional clarifications and restrictions for pragmas defined in Appendix B of the *Reference Manual for the Ada Programming Language*.

### 1.1. Pragma Main

A parameterless library unit procedure can be designated as a main program by including a pragma Main at the end of the specification or body of the unit. This pragma causes the linker to run and create an executable program when the body of this subprogram is coded. Before a unit having a pragma Main can be coded, all units in the transitive with-closure of the unit must be coded.

The pragma Main has arguments that can control the way the linker builds the executable program.

- **Stack\_Size:** This argument takes a static-integer-expression as parameter; it specifies the size, in bytes, of the main task stack. If not specified, the default value is 4K bytes. The value must lie in the range  $256 \dots 2^{31}-1$ .
- **Heap\_Size:** This argument takes a static-integer-expression as parameter; it specifies the size, in bytes, of the heap. The value must lie in the range  $0 \dots 2^{31}-1$ . If not specified, the default value of the Heap\_Size is 64K bytes.

The complete syntax for this pragma is:

```
pragma_main ::= PRAGMA MAIN
              [ ( main_option [ , main_option ] ) ] ;

main_option ::= STACK_SIZE => static_integer_expression |
              HEAP_SIZE => static_integer_expression
```

The pragma Main must appear immediately following the declaration or body of a parameterless library unit procedure.

### 1.2. Pragma Nickname

The pragma Nickname can be used to give a unique string name to a procedure or function besides its normal Ada name. It can be used to disambiguate overloaded procedures or functions in the importing and exporting pragmas defined in subsequent sections.

The pragma Nickname must appear immediately following the declaration for which it is to provide a nickname. It has a single argument, the nickname, which must be a string constant.

For example (pragma Import\_Function is described fully in a subsequent section):

```
function Cat (L: Integer; R: String) return String;
pragma Nickname ("Int-Str-Cat");

function Cat (L: String; R: Integer) return String;
```

```

pragma Nickname ("Str-Int-Cat");

pragma Interface (Assembly, Cat);

pragma Import_Function (Cat, Nickname => "Int-Str-Cat",
                        External => "CAT$INT_STR_CONCAT",
                        Mechanism => (Value, Reference));

pragma Import_Function (Cat, Nickname => "Str-Int-Cat",
                        External => "CAT$STR_INT_CONCAT",
                        Mechanism => (Reference, Value));

```

### 1.3. Importing and Exporting Subprograms

A subprogram written in another language (typically, assembly language) can be called from an Ada program if it is declared with a pragma Interface. The rules for placement of pragma Interface are given in section 13.9 of the *Reference Manual for the Ada Programming Language*. Every interfaced subprogram must have an Vax/VMS cross-compiler defined importing pragma, either the pragma Import\_Procedure or the pragma Import\_Function. These pragmas are used to declare the external name of the subprogram and the parameter passing mechanism for the subprogram call.

A subprogram written in Ada can be made accessible to code written in another language by using an Vax/VMS cross-compiler defined exporting pragma. The effect of such a pragma is to given the subprogram a defined symbolic name that the linker can use when resolving references between object modules.

The importing and exporting pragmas can only be applied to nongeneric procedures and functions.

#### 1.3.1. Subprogram Importing Pragmas

The pragmas Import\_Procedure, Import\_Function, and Import\_Valued\_Procedure are used for importing subprograms. A pragma Interface must precede one of these import pragmas, otherwise, the placement rules for these pragmas are identical to those of the pragma Interface.

The importing pragmas have the form:

```

importing_pragma ::= PRAGMA importing_type
                    ( [ INTERNAL => ] internal_name
                      [ , [ EXTERNAL => ] external_name ]
                      [ [ , [ PARAMETER_TYPES => ] parameter_types ]
                        [ , [ RESULT_TYPE => ] type_mark ] |
                        [ , NICKNAME => string_literal ] ]
                      [ , [ MECHANISM => ] mechanisms ] ) ;

importing_type    ::= IMPORT_PROCEDURE | IMPORT_FUNCTION |
                    IMPORT_VALUED_PROCEDURE

internal_name     ::= identifier |
                    string_literal  -- An operator designator

external_name     ::= identifier | string_literal

parameter_types   ::= ( NULL ) | ( type_mark ( , type_mark ) )

```

```

mechanisms      ::= mechanism_name |
                    ( mechanism_name ( , mechanism_name ) )

mechanism_name  ::= VALUE | REFERENCE | DESCRIPTOR (S)

```

The internal name is the Ada name of the subprogram being interfaced. If there is more than one subprogram in the declarative region, preceeding the importing pragma, then the correct subprogram must be identified using either the argument types (and result type, if a function) or specifying the nickname.

If used to disambiguate an overloaded internal name, the value of the `Parameter_Types` argument consists of a list of type or subtype names, not names of parameters. Each one corresponds, positionally, to a formal parameter in the subprogram's declaration. If the subprogram has no parameters then the list consists of the single word `null`. In the case of a function, the value of the `Result_Type` argument is the name of the type or subtype returned by the function.

The external designator, specified with the `External` parameter, is a character string which is an identifier suitable for the Vax/VMS assembler. If the external designator is not specified, then the internal name is used.

The argument `Mechanism` is required if the subprogram has any parameters. It specifies in a parenthesized list the passing mechanism for each parameter to be passed. The passing mechanism may be one of `Value`, `Reference`, or `Descriptor (S)`. For functions, it is not possible to specify the passing mechanism of the function result; the standard Ada mechanism for the given type of the function result must be used by the interfaced subprogram. If there is one or more parameters, and they all use the same passing mechanism, then an alternate form for the `Mechanism` parameter may be used: Instead of a parenthesized list with an element for each parameter, the single mechanism name (not parenthesized) may be used instead.

Examples:

```

procedure Locate (Source: in String;
                  Target: in String;
                  Index:  out Natural);

pragma Interface (Assembler, Locate);
pragma Import_Procedure
    (Locate, "STR$LOCATE",
     Parameter_Types => (String, String, Natural),
     Mechanism => (Reference, Reference, Value));

function Pwr (I: Integer; N: Integer) return Float;
function Pwr (F: Float; N: Integer) return Float;

pragma Interface (Assembler, Pwr);
pragma Import_Function
    (Internal => Pwr,
     Parameter_Types => (Integer, Integer),
     Result_Type => Float,
     Mechanism => Value,
     External => "MATH$PWR_OF_INTEGER");
pragma Import_Function
    (Internal => Pwr,
     Parameter_Types => (Float, Integer),
     Result_Type => Float,
     Mechanism => Value,
     External => "MATH$PWR_OF_FLOAT");

```

### 1.3.2. Subprogram Exporting Pragmas

The pragmas `Export_Procedure` and `Export_Function` are used to make an Ada subprogram available to external code in a different language by defining a global symbolic name that the linker can use.

An exporting pragma can be given only for subprograms which are library units, or are declared in the outermost declarative part of a library package. An exporting pragma can be placed after a subprogram body only if the subprogram either doesn't have a separate specification or the specification is in the same declarative part as the body. Thus, an exporting pragma can't be applied to the body of a library subprogram which has a separate specification. An exporting pragma cannot be given for a generic library subprogram, or for a subprogram declared in a generic library package.

These pragmas have similar arguments to the importing pragmas, except that it is not possible to specify the parameter passing mechanism. The standard Ada parameter passing mechanisms are chosen. For descriptions of the pragma's arguments, `Internal`, `External`, `Parameter_Types`, `Result_Type`, and `Nickname`, see the preceding section on the importing pragmas.

The full syntax of the pragmas for exporting subprograms is:

```
exporting_pragma ::= PRAGMA exporting_type
                  ( [ INTERNAL => ] internal_name
                    [ , [ EXTERNAL => ] external_name ]
                    [ [ , [ PARAMETER_TYPES => ] parameter_types ]
                    [ , [ RESULT_TYPE => ] type_mark ] |
                    [ , NICKNAME => string_literal ] ) ;

exporting_type   ::= EXPORT_PROCEDURE | EXPORT_FUNCTION

internal_name    ::= identifier |
                  string_literal -- An operator designator

external_name    ::= identifier | string_literal

parameter_types  ::= ( NULL ) | ( type_mark ( , type_mark ) )
```

Examples:

```
procedure Matrix_Multiply
  (A, B: in Matrix; C: out Matrix);

pragma Export_Procedure (Matrix_Multiply);
-- External name is the string "Matrix_Multiply"

function Sin (R: Radians) return Float;
pragma Export_Function
  (Internal => Sin,
   External => "SIN_RADIANS");
-- External name is the string "SIN_RADIANS"
```

### 1.4. Importing and Exporting Objects

Objects can be imported or exported from an Ada unit with the pragmas `Import_Object` and `Export_Object`. The pragma `Import_Object` makes an Ada name reference storage declared and allocated in some external (non-Ada) object module. The pragma `Export_Object` provides an object declared within an

Ada unit with an externally available symbolic name that the linker can use to put together a program with modules written in some other language. In any case, it is the responsibility of the programmer to insure that the internal structure of the object is such that the non-Ada code or data layout matches; such checks cannot be performed by the cross-compiler.

The object to be imported or exported must be a variable declared at the outermost level of a library package specification or body.

The size of the object must be static, thus the type of the object must be one of:

- A scalar type (or subtype).
- An array subtype with static index constraints whose component size is static.
- A simple record type or subtype.

Objects of a private or limited private type can be imported or exported only into the package that declares the type.

Imported objects cannot have an initial value, and thus cannot be:

- constant.
- An access type.
- A task type.
- A record type with discriminants, or with components with default initial expressions, or with components which are access types or task types.

Finally, the object must not be in a generic unit.

The external name specified must be suitable for an identifier in the assembler.

The full syntax for the pragmas `Import_Object` and `Export_Object` is:

```
object_pragma ::= PRAGMA object_pragma_type
                  ( [ INTERNAL => ] identifier
                    [ , [ EXTERNAL => ] string_literal ] ) ;

object_pragma_type ::= IMPORT_OBJECT | EXPORT_OBJECT
```

## 1.5. Pragma `Suppress_All`

This pragma is equivalent to the following sequence of pragmas:

```
pragma Suppress (Access_Check);
pragma Suppress (Discriminant_Check);
pragma Suppress (Division_Check);
pragma Suppress (Elaboration_Check);
pragma Suppress (Index_Check);
pragma Suppress (Length_Check);
pragma Suppress (Overflow_Check);
pragma Suppress (Range_Check);
pragma Suppress (Storage_Check);
```

Note that, just as pragma Suppress, the Suppress\_All pragma cannot prevent the raising of certain exceptions. For example, numeric overflow or dividing by zero is detected by the hardware, which results in the predefined exception Numeric\_Error being raised. Refer to Chapter 7, "Run-Time Organization", for more information.

The pragma Suppress\_All must appear immediately within a declarative part.

## 2. Implementation-Dependent Attributes

There are no implementation-dependent attributes.

## 3. Package System

package System is

```
    type Name is (Dec_Vax);
```

```
    System_Name : constant Name := Dec_Vax;
```

```
    Storage_Unit : constant := 8;
```

```
    Memory_Size : constant := 2 ** 31 - 1;
```

```
    Min_Int : constant := -(2 ** 31);
```

```
    Max_Int : constant := +(2 ** 31) - 1;
```

```
    Max_Digits : constant := 9;
```

```
    Max_Mantissa : constant := 31;
```

```
    Fine_Delta : constant := 2.0 ** (-31);
```

```
    Tick : constant := 1.0E-2;
```

```
    subtype Priority is Integer range 1 .. 254;
```

```
    type Address is private;
```

```
    function "+" (Left : Address; Right : Integer) return Address;
```

```
    function "+" (Left : Integer; Right : Address) return Address;
```

```
    function "-" (Left : Address; Right : Address) return Integer;
```

```
    function "-" (Left : Address; Right : Integer) return Address;
```

```
    function "<" (Left, Right : Address) return Boolean;
```

```
    function "<=" (Left, Right : Address) return Boolean;
```

```
    function ">" (Left, Right : Address) return Boolean;
```

```
    function ">=" (Left, Right : Address) return Boolean;
```

```
    Address_Zero : constant Address;
```

```
    generic
```

```
        type Target is private;
```

```
    function Fetch_From_Address (A : Address) return Target;
```

```
    generic
```

```
        type Target is private;
```

```
    procedure Assign_To_Address (A : Address; T : Target);
```



```

type Type_Class is (Type_Class_Enumeration, Type_Class_Integer,
                    Type_Class_Fixed_Point, Type_Class_Floating_Point,
                    Type_Class_Array, Type_Class_Record,
Type_Class_Access,
                    Type_Class_Task, Type_Class_Address);
type F_Float is digits 6;
type D_Float is digits 9;

type Ast_Handler is limited private;
No_Ast_Handler : constant Ast_Handler;

type Bit_Array is array (Integer range <>) of Boolean;
pragma Pack (Bit_Array);

subtype Bit_Array_8 is Bit_Array (0 .. 7);
subtype Bit_Array_16 is Bit_Array (0 .. 15);
subtype Bit_Array_32 is Bit_Array (0 .. 31);
subtype Bit_Array_64 is Bit_Array (0 .. 63);

type Unsigned_Byte is range 0 .. 255;

function "not" (Left : Unsigned_Byte) return Unsigned_Byte;
function "and" (Left, Right : Unsigned_Byte) return Unsigned_Byte;
function "or" (Left, Right : Unsigned_Byte) return Unsigned_Byte;
function "xor" (Left, Right : Unsigned_Byte) return Unsigned_Byte;

function To_Unsigned_Byte (Left : Bit_Array_8) return Unsigned_Byte;
function To_Bit_Array_8 (Left : Unsigned_Byte) return Bit_Array_8;

type Unsigned_Byte_Array is array (Integer range <>) of Unsigned_Byte;
pragma Pack (Unsigned_Byte_Array);

type Unsigned_Word is range 0 .. 65535;

function "not" (Left : Unsigned_Word) return Unsigned_Word;
function "and" (Left, Right : Unsigned_Word) return Unsigned_Word;
function "or" (Left, Right : Unsigned_Word) return Unsigned_Word;
function "xor" (Left, Right : Unsigned_Word) return Unsigned_Word;

function To_Unsigned_Word (Left : Bit_Array_16) return Unsigned_Word;
function To_Bit_Array_16 (Left : Unsigned_Word) return Bit_Array_16;

type Unsigned_Word_Array is array (Integer range <>) of Unsigned_Word;
pragma Pack (Unsigned_Word_Array);

type Unsigned_Longword is range Min_Int .. Max_Int;

function "not" (Left : Unsigned_Longword) return Unsigned_Longword;
function "and" (Left, Right : Unsigned_Longword) return Unsigned_Longword;
function "or" (Left, Right : Unsigned_Longword) return Unsigned_Longword;
function "xor" (Left, Right : Unsigned_Longword) return Unsigned_Longword;

function To_Unsigned_Longword
    (Left : Bit_Array_32) return Unsigned_Longword;
function To_Bit_Array_32 (Left : Unsigned_Longword) return Bit_Array_32;

```

```

type Unsigned_Longword_Array is
    array (Integer range <>) of Unsigned_Longword;

type Unsigned_Quadword is
    record
        L0 : Unsigned_Longword;
        L1 : Unsigned_Longword;
    end record;

function "not" (Left : Unsigned_Quadword) return Unsigned_Quadword;
function "and" (Left, Right : Unsigned_Quadword) return Unsigned_Quadword;
function "or" (Left, Right : Unsigned_Quadword) return Unsigned_Quadword;
function "xor" (Left, Right : Unsigned_Quadword) return Unsigned_Quadword;

function To_Unsigned_Quadword
    (Left : Bit_Array_64) return Unsigned_Quadword;
function To_Bit_Array_64 (Left : Unsigned_Quadword) return Bit_Array_64;

type Unsigned_Quadword_Array is
    array (Integer range <>) of Unsigned_Quadword;

function To_Address (X : Integer) return Address;
function To_Address (X : Unsigned_Longword) return Address;

function To_Integer (X : Address) return Integer;
function To_Unsigned_Longword (X : Address) return Unsigned_Longword;

subtype Unsigned_1 is Unsigned_Longword range 0 .. 2 ** 1 - 1;
subtype Unsigned_2 is Unsigned_Longword range 0 .. 2 ** 2 - 1;
subtype Unsigned_3 is Unsigned_Longword range 0 .. 2 ** 3 - 1;
subtype Unsigned_4 is Unsigned_Longword range 0 .. 2 ** 4 - 1;
subtype Unsigned_5 is Unsigned_Longword range 0 .. 2 ** 5 - 1;
subtype Unsigned_6 is Unsigned_Longword range 0 .. 2 ** 6 - 1;
subtype Unsigned_7 is Unsigned_Longword range 0 .. 2 ** 7 - 1;
subtype Unsigned_8 is Unsigned_Longword range 0 .. 2 ** 8 - 1;
subtype Unsigned_9 is Unsigned_Longword range 0 .. 2 ** 9 - 1;
subtype Unsigned_10 is Unsigned_Longword range 0 .. 2 ** 10 - 1;
subtype Unsigned_11 is Unsigned_Longword range 0 .. 2 ** 11 - 1;
subtype Unsigned_12 is Unsigned_Longword range 0 .. 2 ** 12 - 1;
subtype Unsigned_13 is Unsigned_Longword range 0 .. 2 ** 13 - 1;
subtype Unsigned_14 is Unsigned_Longword range 0 .. 2 ** 14 - 1;
subtype Unsigned_15 is Unsigned_Longword range 0 .. 2 ** 15 - 1;
subtype Unsigned_16 is Unsigned_Longword range 0 .. 2 ** 16 - 1;
subtype Unsigned_17 is Unsigned_Longword range 0 .. 2 ** 17 - 1;
subtype Unsigned_18 is Unsigned_Longword range 0 .. 2 ** 18 - 1;
subtype Unsigned_19 is Unsigned_Longword range 0 .. 2 ** 19 - 1;
subtype Unsigned_20 is Unsigned_Longword range 0 .. 2 ** 20 - 1;
subtype Unsigned_21 is Unsigned_Longword range 0 .. 2 ** 21 - 1;
subtype Unsigned_22 is Unsigned_Longword range 0 .. 2 ** 22 - 1;
subtype Unsigned_23 is Unsigned_Longword range 0 .. 2 ** 23 - 1;
subtype Unsigned_24 is Unsigned_Longword range 0 .. 2 ** 24 - 1;
subtype Unsigned_25 is Unsigned_Longword range 0 .. 2 ** 25 - 1;
subtype Unsigned_26 is Unsigned_Longword range 0 .. 2 ** 26 - 1;
subtype Unsigned_27 is Unsigned_Longword range 0 .. 2 ** 27 - 1;
subtype Unsigned_28 is Unsigned_Longword range 0 .. 2 ** 28 - 1;
subtype Unsigned_29 is Unsigned_Longword range 0 .. 2 ** 29 - 1;

```

```
    subtype Unsigned_30 is Unsigned_Longword range 0 .. 2 ** 30 - 1;  
    subtype Unsigned_31 is Unsigned_Longword range 0 .. 2 ** 31 - 1;  
  
private  
    ...  
end System;
```

## **4. Restrictions on Representation Clauses**

### **4.1. Length clauses**

Size specifications are allowed only on discrete types that are not derived types. There are no restrictions on specifications of collection size or task activation size. *Small* may be specified for a fixed point type so long as it is a power of two that does not exceed the given delta and still permits representation of all necessary values, and so long as the fixed point type is not a derived type.

### **4.2. Enumeration Representation Clauses**

Enumeration representation clauses are not permitted on derived enumeration types.

### **4.3. Record Representation Clauses**

A record field can consist of any number of bits between 1 and 32 inclusive; otherwise it must be an integral number of 8-bit bytes. There are no other restrictions on record field specifications.

### **4.4. Address Clauses**

Address clauses are not supported.

## **5. Names denoting implementation-dependent components**

There are no implementation dependent components that can be named in representation clauses.

## **6. Interpretation of expressions that appear in address clauses**

Address clauses are not supported.

## **7. Unchecked conversion**

The target type of an unchecked conversion cannot be an unconstrained array type or an unconstrained discriminated type.

## 8. Implementation dependent characteristics of the input output packages

### 8.1. The generic packages `Sequential_io` and `Direct_Io`

`Direct_Io` may only be instantiated with constrained types.

The implementation of `Sequential_Io` creates VMS RMS variable-length-record record-oriented files for unconstrained types and fixed-length-record record-oriented files for constrained types. `Direct_Io` always creates fixed-length-record record-oriented files. All files are created with the carriage return carriage control attribute and use the VMS RMS sequential organization. There is a one-to-one correspondence between internal file elements and external file records. Thus, `Element_Type`' size is limited to 32767 bytes by the VMS RMS I/O system. The implementation can read files in these formats only.

The integer type `Direct_Io.Count` is defined to have an upper bound of `Integer'last`. However, because of limitations of the VMS RMS I/O system, the actual limit is  $512 * \text{integer'last} / \text{element\_type' size}$ .

The `Form` parameter is ignored on `Open` and `Create` calls.

Input and output files are not buffered beyond that provided by VMS RMS.

The implementation does not support having the same external file open for both input and output via two different internal files or having the same external file open for writing via two different internal files. Temporary files are named, but a single temporary file cannot be shared by two or more different internal files.

`Use_Error` is raised when a `Create` is attempted with the mode `In_File`. `Read` does not perform the extra checks that might raise `Data_Error` for values inappropriate for the element type.

When creating a file, the default protections of the containing directory are used. The implementation does not examine the previous version of the file to use its protection.

### 8.2. The package `Text_Io`

The integer type `Text_Io.Count` is defined to have an upper bound of 1\_000\_000\_000. However, this isn't practical: The VMS RMS I/O system limits record length to 32767 bytes for disk files and 255 bytes for terminals. `Text_Io.Field` has the same range as type `Standard.Natural`.

The implementation of `Text_Io` creates VMS RMS variable-length-record record-oriented files with carriage return carriage control attributes. VMS RMS stream files are not created for output files. In these files, each record corresponds to one `Text_Io` line. A page terminator is represented by an `ASCII_FF` either as the first character or the last character of a record. `ASCII_FF` characters embedded within records except as the first or last character of the record are not page terminators. The page terminator at the end of the file is implicit, there is no `ASCII_FF` in the file to represent it. `Text_Io` can read either record-oriented files or stream-oriented files.

For the file `Standard_Input`, if the logical name `ADASINPUT` is defined then it is opened, otherwise `SYSSINPUT` is opened. If `SYSSINPUT` isn't defined (if, for example, the program was spawned by the `RUN` command) then `NULL:` is used. (`NULL:` is the standard VMS file name for the bit-bucket, when used as an input file it returns end-of-file immediately.)

For the file `Standard_Output`, if a logical name `ADASOUTPUT` is defined then it will be used as the name of the file to be created, otherwise `SYSSOUTPUT` is used.

The `FORM` parameter is ignored on `Open` and `Create` calls.

Output to a terminal is not buffered. Output to files is buffered, but as part of (normal or abnormal) program termination the Ada runtimes call Text\_Io in order to flush the buffers of all files so they can be closed without the loss of information.

The implementation of Text\_Io does not support having the same external file open for both input and output via two different internal files (Text\_Io.File\_Type) or having the same external file open for writing via two different internal files.

When creating a file, the default protections of the containing directory are used. Text\_Io does not examine the previous version of the file to use its protection.

## 9. Standard package

package Standard is

```
type *Universal_Integer* is [universal_integer];
type *Universal_Real* is [universal_real];
type *Universal_Fixed* is [universal_fixed];

type Boolean is (False, True);

type Integer is range -2147483648 .. 2147483647;
type Short_Short_Integer is range -128 .. 127;
type Short_Integer is range -32768 .. 32767;

type Float is digits 6
               range -1.70141173319264E+38 .. 1.70141173319264E+38;
type Long_Float is digits 9
               range -1.70141183460469E+38 .. 1.70141183460469E+38;

type Duration is delta 6.103515625000000E-05
               range -1.310720000000000E+05 .. 1.31071999938965E+05;

subtype Natural is Integer range 0 .. 2147483647;
subtype Positive is Integer range 1 .. 2147483647;

type String is array (Positive range <>) of Character;
pragma Pack (String);

package Ascii is
    ...
end Ascii;

Constraint_Error : exception;
Numeric_Error : exception;
Storage_Error : exception;
Tasking_Error : exception;
Program_Error : exception;

type Character is ...

end Standard;
```

APPENDIX C  
TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

<u>Name and Meaning</u>	<u>Value</u>
\$BIG_ID1 Identifier the size of the maximum input line length with varying last character.	(1..253 => 'A', 254 => '1')
\$BIG_ID2 Identifier the size of the maximum input line length with varying last character.	(1..253 => 'A', 254 => '2')
\$BIG_ID3 Identifier the size of the maximum input line length with varying middle character.	(1..126 => 'A', 127 => '3', 128..254 => 'A')
\$BIG_ID4 Identifier the size of the maximum input line length with varying middle character.	(1..126 => 'A', 127 => '4', 128..254 => 'A')
\$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	(1..251 => '0', 252..254 => "298")

# TEST PARAMETERS

<u>Name and Meaning</u>	<u>Value</u>
<b>\$BIG_REAL_LIT</b> A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	(1..248 => '0', 249...254 => "69.0E1")
<b>\$BIG_STRING1</b> A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	(1..127 => 'A')
<b>\$BIG_STRING2</b> A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	(1..126 => 'A', 127 => '1')
<b>\$BLANKS</b> A sequence of blanks twenty characters less than the size of the maximum line length.	(1..234 => ' ')
<b>\$COUNT_LAST</b> A universal integer literal whose value is TEXT_IO.COUNT'LAST.	1000000000
<b>\$FIELD_LAST</b> A universal integer literal whose value is TEXT_IO.FIELD'LAST.	2147483647
<b>\$FILE_NAME_WITH_BAD_CHARS</b> An external file name that either contains invalid characters or is too long.	BAD_CHARACTERS&<>=
<b>\$FILE_NAME_WITH_WILD_CARD_CHAR</b> An external file name that either contains a wild card character or is too long.	WILDCARDS*
<b>\$GREATER_THAN_DURATION</b> A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	0.0

## TEST PARAMETERS

<u>Name and Meaning</u>	<u>Value</u>
\$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	2.0E05
\$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	BAD_CHARACTERS&<>=
\$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	STRING'(1..100=>'A')&STRING'(1..100=>'A') &STRING'(1..100=>'A')
\$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-2147483648
\$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	2147483647
\$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	2147483648
\$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	0.0
\$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-2.0E05
\$MAX_DIGITS Maximum digits supported for floating-point types.	9
\$MAX_IN_LEN Maximum input line length permitted by the implementation.	254
\$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
\$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2147483648



# TEST PARAMETERS

<u>Name and Meaning</u>	<u>Value</u>
<p><b>\$MAX_LEN_INT_BASED_LITERAL</b>            A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.</p>	<p>(1..2 =&gt; "2:", 3..251 =&gt; '0',            252..254 =&gt; "11:")</p>
<p><b>\$MAX_LEN_REAL_BASED_LITERAL</b>            A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.</p>	<p>(1..3 =&gt; "16:", 4..250 =&gt; '0',            251..254 =&gt; "F.E:")</p>
<p><b>\$MAX_STRING_LITERAL</b>            A string literal of size MAX_IN_LEN, including the quote characters.</p>	<p>(1 =&gt; '"', 2..253 =&gt; 'A', 254 =&gt; '"')</p>
<p><b>\$MIN_INT</b>            A universal integer literal whose value is SYSTEM.MIN_INT.</p>	<p>-2147483648</p>
<p><b>\$NAME</b>            A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.</p>	<p>SHORT_SHORT_INTEGER</p>
<p><b>\$NEG_BASED_INT</b>            A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.</p>	<p>16#FFFFFFFFFFFFFFFFFE#</p>

## APPENDIX D

### WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 27 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form "AI-ddddd" is to an Ada Commentary.

- . B28003A: A basic declaration (line 36) incorrectly follows a later declaration.
- . E28005C: This test requires that "PRAGMA LIST (ON);" not appear in a listing that has been suspended by a previous "PRAGMA LIST (OFF);"; the Ada Standard is not clear on this point, and the matter will be reviewed by the AJPO.
- . C34004A: The expression in line 168 yields a value outside the range of the target type T, but there is no handler for CONSTRAINT\_ERROR.
- . C35502P: The equality operators in lines 62 and 69 should be inequality operators.
- . A35902C: The assignment in line 17 of the nominal upper bound of a fixed-point type to an object raises CONSTRAINT\_ERROR, for that value lies outside of the actual range of the type.
- . C35904A: The elaboration of the fixed-point subtype on line 28 wrongly raises CONSTRAINT\_ERROR, because its upper bound exceeds that of the type.
- . C35904B: The subtype declaration that is expected to raise CONSTRAINT\_ERROR when its compatibility is checked against that of various types passed as actual generic parameters, may, in fact, raise NUMERIC\_ERROR or CONSTRAINT\_ERROR for reasons not anticipated by the test.

## WITHDRAWN TESTS

- . C35A03E and C35A03R: These tests assume that attribute 'MANTISSA returns 0 when applied to a fixed-point type with a null range, but the Ada Standard does not support this assumption.
- . C37213H: The subtype declaration of SCONS in line 100 is incorrectly expected to raise an exception when elaborated.
- . C37213J: The aggregate in line 451 incorrectly raises CONSTRAINT\_ERROR.
- . C37215C, C37215E, C37215G, and C37215H: Various discriminant constraints are incorrectly expected to be incompatible with type CONS.
- . C38102C: The fixed-point conversion on line 23 wrongly raises CONSTRAINT\_ERROR.
- . C41402A: The attribute 'STORAGE\_SIZE is incorrectly applied to an object of an access type.
- . C45332A: The test expects that either an expression in line 52 will raise an exception or else MACHINE\_OVERFLOW is FALSE. However, an implementation may evaluate the expression correctly using a type with a wider range than the base type of the operands, and MACHINE\_OVERFLOW may still be TRUE.
- . C45614C: The function call of IDENT\_INT in line 15 uses an argument of the wrong type.
- . A74106C, C85018B, C87B04B, and CC1311B: A bound specified in a fixed-point subtype declaration lies outside of that calculated for the base type, raising CONSTRAINT\_ERROR. Errors of this sort occur at lines 37 & 59, 142 & 143, 16 & 48, and 252 & 253 of the four tests, respectively.
- . BC3105A: Lines 159 through 168 expect error messages, but these lines are correct Ada.
- . AD1A01A: The declaration of subtype SINT3 raises CONSTRAINT\_ERROR for implementations which select INT'SIZE to be 16 or greater.
- . CE2401H: The record aggregates in lines 105 and 117 contain the wrong values.
- . CE3208A: This test expects that an attempt to open the default output file (after it was closed) with mode IN\_FILE raises NAME\_ERROR or USE\_ERROR; by Commentary AI-00048, MODE\_ERROR should be raised.